



Correctness by Construction: A Manifesto for High-Integrity Software

Martin Croxford and Dr. Roderick Chapman
Praxis High Integrity Systems

High-integrity software systems are often so large that conventional development processes cannot get anywhere near achieving tolerable defect rates. This article presents an approach that has delivered software with very low defect rates cost-effectively. We describe the technical details of the approach and the results achieved, and discuss how to overcome barriers to adopting such best practice approaches. We conclude by observing that where such approaches are compatible and can be deployed in combination, we have the opportunity to realize the extremely low defect rates needed for high integrity software composed of many million lines of code.

The National Institute of Standards and Technology (NIST) reported in 2002 that low quality software costs the U.S. economy \$60 billion per year [1]. According to the aptly named “Chaos Report,” only one quarter of software projects are judged a success [2]. Software defects are accepted as inevitable by both the software industry and the long-suffering user community. In any other engineering discipline, this defect rate would be unacceptable. But when safety and security are at stake, the extent of current software vulnerability is unsustainable.

Recent research on this issue has been conducted on behalf of the National Cyber Security Partnership, formed in 2003 in response to the White House National Strategy to Secure Cyberspace [3]. The partnership’s Secure Software Task Force report states the following:

Software security vulnerabilities are often caused by defective specification, design, and implementation. Unfortunately today, common development practices can often leave numerous defects and resulting vulnerabilities in the complex artifact that is delivered software. To have a secure U.S. cyber infrastructure, the supporting software must contain few, if any, vulnerabilities. [4]

The report goes on to recommend adoption of software development processes that can measurably reduce software specification, design, and implementation defects. It identifies three software engineering practices as examples that satisfy this recommendation. This article describes one of these examples, *Correctness by Construction* (CbyC), which

originates from Praxis High Integrity Systems.

Maturity of Approach

The CbyC approach has two primary goals: to deliver software with defect rates an order of magnitude lower than current best commercial practices in a cost-effective manner, and to deliver durable software that is resilient to change throughout its life cycle.

Elements of the CbyC approach have been used for more than 15 years to produce software with very low defects mainly for safety-critical applications, but more recently for security-critical applications. The approach has evolved over time and now applies to the entire systems development life cycle, from validation of the concepts of operation to preserving correctness properties during long-term maintenance.

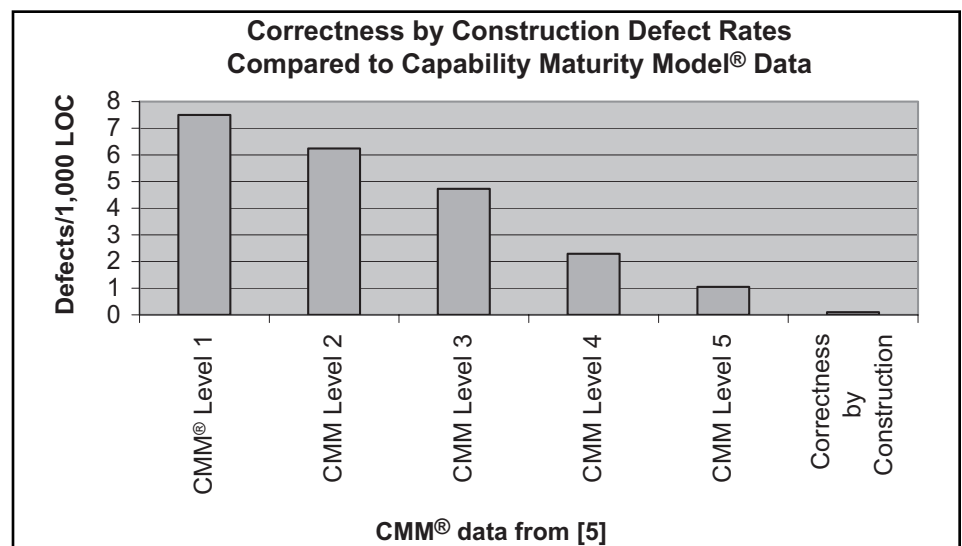
CbyC has delivered software with defect rates of less than 0.1 defects/1,000 source lines of code (SLOC) with good productivity: up to around 30 LOC per

day. The achieved defect rates compare very favorably with defect rates reported by Capability Maturity Model® Level 5 organizations of 1 defect/1,000 LOC [5]. The comparative rates are shown in Figure 1. It is, of course, true that other approaches have also succeeded in delivering similarly low defect rates, however, it is rare to also deliver good productivity (since low defect rates are often the result of extensive, expensive debugging and testing).

As well as realizing low defect rates, the CbyC approach has also proved to be highly cost-effective during both development and maintenance. Metrics for five fully deployed projects are shown in Figure 2 (see page 7).

Given that CbyC and other best-practice approaches cited in the National Cyber Security Summit Task Force report [4] have been used so successfully for a number of years, you may ask: Why are these approaches not in more widespread use, especially where high levels of assurance are required?

Figure 1: *Correctness by Construction Defect Rates Comparison*



* Capability Maturity Model is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Before considering the barriers to adoption of best practices, it is necessary to examine the nature of CbyC. A more detailed white paper on CbyC [6] is freely available from the authors.

Fundamental Principles

The primary goals of very low defect rate and very high resilience to change are realized in CbyC by two fundamental principles: to make it difficult to introduce errors in the first place, and to detect and remove any errors that do occur as early as possible after introduction.

The key to implementing these principles is to introduce sufficient precision at each step of the software development to enable reasoning about the correctness of that step – reasoning in the sense that an argument for correctness can be established either by review or using tool support. The aim is to demonstrate or argue the software correctness in terms of the manner in which it has been produced (*by construction*) rather than just by observing operational behavior.

It is the use of precision that differentiates approaches such as CbyC from others in common use. Typically, software development approaches endure a lack of precision that makes it very easy to introduce errors, and very hard to find those errors early. Evidence for this may be found in the common tendency for development life cycles to migrate to an often-repeating *code-test-debug* phase, which can lead to severe cost and timescale overruns.

Conversely, the rigor and precision of the CbyC approach means that the requirements are more likely to be correct, the system is more likely to be the correct system to meet the requirements, the implementation is more likely to be defect-free, and upgrades are more likely to retain the original correctness properties.

Achieving the Fundamental Principles

The principles of making it difficult to introduce defects and making it easy to detect and remove errors early are achieved by a combination of the following six strategies:

1. **Using a sound, formal notation for all deliverables.** For example, using Z [7] for writing specifications so it is impossible to be ambiguous, or using SPARK [8] to write the code so it is impossible to introduce errors such as buffer overflows.
2. **Using strong, tool-supported methods to validate each deliverable.** For example, carrying out proofs of formal

specifications and static analysis of code. This is only possible where formal notations are used (strategy No. 1).

3. **Carrying out small steps and validating the deliverable from each step.** For example, developing a software specification as an elaboration of the user requirements, and checking that it is correct before writing code. For example, building the system in small increments and checking that each increment behaves correctly.
4. **Saying things only once.** For example, by producing a software specification that says what the software will do and a design that says how it will be structured. The design does not repeat any information in the specification, and the two can be produced in parallel.
5. **Designing software that is easy to validate.** For example, writing simple code that directly reflects the specification, and testing it using tests derived systematically from that specification.
6. **Doing the hard things first.** For example, by producing early prototypes to test out difficult design issues or key user interfaces.

These six principles are not in themselves difficult to apply, and may even appear obvious. However, in the authors' experience, many software development projects fail to deliver against many, if any, of these principles.

Requirements Engineering

At the requirements step (a source of half of project failures [2]), a clear distinction is made between user requirements, system specifications, and domain knowledge. CbyC uses *satisfaction arguments* to show that each user requirement can be satisfied by an appropriate combination of system specification and domain knowledge. The emphasis on domain knowledge is key – half of all requirements errors are related to domain [9] – yet the vast majority of requirements processes do not explicitly address issues in the domain.

Formal Specification and Design

Using mathematical (or formal) methods and notations to define the specification and high-level design provide a precise description of behavior and a precise model of its characteristics. This enables using tools to verify that the design meets its specification and that the specification meets its requirements.

Example languages used for formal specification in CbyC projects include Z

and Communicating Sequential Processes [10].

Development

The CbyC approach applies rigor to all software development phases, including detailed design, implementation, and verification. As a result, static analysis tools can be used to produce evidence of correctness and completeness.

CbyC defines a software design methodology based on information flow that can be expressed using an unambiguous notation. This notation is contract-based, i.e., it is used to define both the abstract state and the information relationships across the software modules.

For the coding phase, the CbyC approach recommends using languages and tools that are most appropriate for the task at hand. Validation requirements play a large role in this choice: The selected languages must be amenable to verification and analysis so that the required evidence of correctness can be generated effectively. For high-integrity software modules, the SPARK programming language is especially suitable owing to its rigorous and unambiguous semantics.

Using mathematically verifiable programming languages such as SPARK opens the way for static analysis tools to provide proofs for absence of common runtime errors such as buffer overflows and using uninitialized variables. Being able to prove absence of runtime errors, rather than discovering a subset of them by testing, is critical to the achievement of very low defect rates.

Note that other programming languages have been used for CbyC projects, and that CbyC projects often have an element of mixed-language implementation. For example, C, C++, Structured Query Language (SQL), Ada '83 and Ada '95 have been used. However, such languages are intrinsically unsuitable for deep static analysis and are only ever used for the non-critical parts of the implementation.

Results

Experience from a wide variety of projects has confirmed that CbyC is both effective and economical due to the following:

1. Defects are removed early in the process when changes are cheap. Testing becomes a confirmation that the software works, rather than the point at which it must be debugged.
2. Evidence needed for safety or security certification is produced naturally as a byproduct of the process.
3. Early iterations produce software that

carries out useful functions and builds confidence in the project.

Figure 2 shows results from three safety-critical and two security-critical projects that have used elements of the CbyC approach. For all of these projects, the reported productivity figures are for the whole life cycle, from requirements to delivery.

The Ship/Helicopter Operating Limits Information System [11] was developed in 1997 and was the first project to be developed to the full degree of rigor required by the United Kingdom (UK) Ministry of Defence (MoD), Defence Standard 00-55 [12] at the highest safety integrity level.

The certification authority system to support the Multimedia Office Server (MULTOS) smart card operating system developed by Mondex International [13] was developed to the standards of the Information Technology Security Evaluation Criteria (ITSEC) Level E6¹, roughly equivalent to Common Criteria Evaluation Assurance Level (EAL) 7. The system had an operational defect rate of 0.04 defects/KLOC, yet was developed at a productivity of almost 30 LOC per day (three times typical industry figures).

CbyC was used in 2003 to develop a demonstrator biometrics system for the National Security Agency (NSA), aimed at showing that it is possible to produce cost-effective, high-quality, low-defect software conforming to the Common Criteria EAL 5 and above [14]. The software was subjected to rigorous independent reliability testing that identified zero defects and was developed at a productivity of almost 40 LOC/day.

These and other similar projects have demonstrated that the rigorous techniques employed by CbyC such as formal methods and proofs should no longer be viewed as belonging solely to academia, but can be used confidently and effectively in the commercial sector.

Barriers to Adoption

Earlier, the question asked was why best practices such as CbyC and others referenced by [4] are not in widespread use. The authors contend that there are two kinds of barriers to the adoption of best practices.

First, there is often a cultural mindset or awareness barrier. Many individuals and organizations do not recognize or believe that it is possible to develop software that is low-defect, high-integrity, and cost-effective. This may simply be an awareness

Project	Year	Size (SLOC)	Whole Life-Cycle Productivity (SLOC/day)	Defects (/1,000 SLOC)
CDIS ¹	1992	197,000	12.7	0.75
SHOLIS ²	1997	27,000	7.0	0.22
MULTOS CA ³	1999	100,000	28.0	0.04
A ⁴	2001	39,000	11.0	0.05
NSA ⁵	2003	10,000	38.0	0

Notes

¹ Real-time air traffic information system at the London Terminal Control Centre.

² Ship/Helicopter Operating Limits Information System developed to UK MoD Defence Standard 00-55 Safety Integrity Level 4 (highest).

³ Certification authority for smart card operating system maintained by Mastercard.

⁴ A UK military stores management system.

⁵ NSA Tokeneer ID Station demonstrator biometrics system.

Figure 2: *Correctness By Construction Project Metrics*

issue, in principle readily addressed by articles such as this. Or there may be a view that such best practices *could never work here* for a combination of reasons. These reasons are likely to include perceived capability of the staff, belief about applicability to the organization's product or process, prevalence of legacy software that is viewed as inherently inappropriate for such approaches, or concern about the disruption and cost of introducing new approaches.

Second, where the need for improvement is acknowledged and considered achievable, there are usually practical barriers to overcome such as how to acquire the necessary capability or expertise, and how to introduce the changes necessary to make the improvements.

Overcoming the Barriers

The barriers mentioned above are reasonable and commonplace, but not insurmountable. Overcoming them requires effort from suppliers, procurers, and regulators and involvement at the individual, project, and organizational level. Typically, strong motivation and leadership will be required at a senior management level where the costs to the business of poor quality (high defects, low productivity, and lack of resilience to change) are most likely to be experienced.

The authors have worked with a number of organizations to overcome these barriers. For example, the MULTOS system was delivered to Mondex International, along with three weeks training in the techniques used to develop it, and three weeks of part-time mentoring. Mondex has since successfully maintained the system – to the same development

standards – with no further support from Praxis. The NSA system was successfully adapted by summer interns during a 12-week placement after minimal training in the techniques used to develop it.

The key to successful adoption of CbyC is the adoption of an engineering mindset. In particular, decisions on process, methods, and tools for software development need to be premised on the basis of logic and precision (for example, by asking, “How does this choice help me meet one of the six strategies of CbyC?”), rather than on fashion (characterized by questions such as, “How many developers already know this particular technology?”).

Procurers have a role in overcoming barriers to best practices by demanding low defects. Regulation also has a role to play in requiring best practices; this is already happening within the security sector, for example Common Criteria EAL 5 and above, and within the safety sector, particularly in Europe, for example in the UK Civil Aviation Authority regulatory objectives for software [15] and the UK MoD safety standard 00-55 [12].

Maximizing the Benefit

Given the massive size of many software systems – some of which need to be high integrity – even a defect rate of 0.04 defect per KLOC may result in an unacceptably high number of faults. To address this, we need to employ a combination of compatible defect-prevention approaches.

One of the other identified approaches in the Secure Software Task Force report is the Team Software ProcessSM (TSPSM) and Personal Software ProcessSM

SM Team Software Process, Personal Software Process, TSP, and PSP are service marks of Carnegie Mellon University.

(PSPSM) from the Software Engineering Institute [16]. Since the focus of TSP/PSP is on improving the professional culture and working practices of individuals, teams, and management, and hence is largely independent of languages, tools, and methodologies that are used, the deployment of CbyC within an environment such as TSP/PSP is highly feasible and has already been demonstrated: A CbyC practitioner's results at a recent PSP training course were both defect-free and first to be completed. Given that the TSP/PSP approach has also demonstrated a very low defect rate, the combination of these approaches offers the best opportunity to realize the orders of magnitude reduction in a defect rate that are needed for a multi-million LOC high-integrity software subsystem.

Conclusions

Critical software subsystems are now large enough such that conventional development processes cannot get anywhere near reducing defect rates to tolerable levels.

A mature approach based on applying rigor and precision to each phase of the life cycle has demonstrated over the past 15 years that major improvements in defect rate are attainable while maintaining productivity levels and overall cost-effectiveness.

Where such compatible approaches can be deployed in combination, we can at last see extremely low defect rates needed for high-integrity software composed of many million lines of code. ♦

Acknowledgements

The authors acknowledge contributions from Brian Dobbing, Peter Amey, and Anthony Hall of Praxis High Integrity Systems.

References

1. Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing. Ed. Dr. Gregory Tasse. RTI Project No. 7007.011. Washington, D.C.: National Institute of Standards and Technology, May 2002 <www.nist.gov/msd/sima/sw_testing_rpt.pdf>.
2. Standish Group International. The Chaos Report. West Yarmouth, MA: Standish Group International, 2003 <www.standishgroup.com>.
3. National Cyber Security Partnership. "About the National Cyber Security Partnership." Washington, D.C.: NCSP, 18 Mar. 2004 <www.cyberpartnership.org/about-overview.html>.
4. National Cyber Security Task Force. "Improving Security Across the Software Development Life Cycle." Washington, D.C.: National Cyber Security Partnership, 1 Apr. 2004 <www.cyberpartnership.org/init-soft.html>.
5. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison-Wesley, 2000.
6. Praxis High Integrity Systems. "Correctness by Construction: A White Paper." Issue 1.2, Jan. 2005. Please contact the authors for a copy of this paper.
7. Spivey, J.M. The Z Notation: A Reference Manual. 2nd ed. Prentice-Hall, 1992.
8. Barnes, J. High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley, 2003.
9. Hooks, Ivy F., and Kristin A. Farry. Customer Centered Products: Creating Successful Products Through Smart Requirements Management. 1st ed. New York: American Management Assoc., 11 Sept. 2000.
10. Hoare, C.A.R. Communicating Sequential Processes. Prentice-Hall, 1985.
11. King S., J. Hammond, R. Chapman, and A. Pryor. "Is Proof More Cost-Effective Than Testing?" IEEE Transactions on Software Engineering 26.8 (Aug. 2000) <www.praxis-his.com/pdfs/cost_effective_proof.pdf>.
12. United Kingdom Ministry of Defence. "Def. Stan. 00-55." Requirements for Safety Related Software in Defense Equipment Issue 2, Aug. 1997.
13. Hall, A., and R. Chapman R. "Correctness by Construction: Developing a Commercial Secure System." IEEE Software Jan./Feb. 2002 <www.praxis-his.com/pdfs/c_by_c_secure_system.pdf>.
14. National Security Agency. Fourth Annual High Confidence Software and Systems Conference Proceedings. Washington, D.C.: NSA, Apr. 2004.
15. United Kingdom Civil Aviation Authority. "CAP 670, Air Traffic Services Safety Requirements. Amendment 3." UKCAA, Sept. 1999.
16. Humphrey, W. Introduction to the Team Software Process. Addison-Wesley, 2000.

Note

1. Information about ITSEC and the Common Criteria can be found at <www.cesg.gov.uk/site/iacs/index.cfm>.

About the Authors



Martin Croxford is associate director for security with Praxis High Integrity Systems, a United Kingdom-based systems engineering company specializing in mission-critical systems. He is a chartered engineer with 15 years experience in the software industry. Croxford has worked on software development projects in a range of organizations, and as a software development manager has used Correctness by Construction to successfully deliver a multi-million dollar security-critical system.

Praxis High Integrity Systems
20 Manvers ST
BATH BA1 1PX
UK
Phone: (44) 1225-823794
Fax: (44) 1225-469006
E-mail: martin.croxford@praxis-his.com



Roderick Chapman, Ph.D., is product manager of SPARK with Praxis High Integrity Systems, specializing in the development of programming languages and static analysis tools for high integrity systems. He is a chartered engineer with more than a decade of experience in high integrity real-time systems. Chapman is internationally renowned for his work on verification of correctness properties of high integrity software. He has a Doctor of Philosophy in computer science.

Praxis High Integrity Systems
20 Manvers ST
BATH BA1 1PX
UK
Phone: (44) 1225-823763
Fax: (44) 1225-469006
E-mail: rod.chapman@praxis-his.com